CMSC 631 – Program Analysis and Understanding
Spring 2009

Type Systems

The Need for a Type System

- Consider the (untyped) lambda calculus
  - false = λx.λy.x
  - 0 (Scott) = λx.λy.x
- Everything is encoded as a function
  - So we can easily misuse combinators
    - false 0   if 0 then ...   etc...
  - This is no better than assembly language!

What is a Type System?

- A *type system* is some mechanism for distinguishing good programs from bad
  - Good programs = well typed
  - Bad programs = ill typed or not typable
- Examples:
  - 0 + 1   // well typed
  - false 0   // ill-typed: can't apply a boolean
  - 1 + (if true then 0 else false) // ill-typed: can't add boolean to integer

A Definition of Type Systems

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

– Benjamin Pierce, *Types and Programming Languages*

Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$
  - Functions include the type of their argument
  - We don't really need this, but it will come in handy
- $t ::= \text{int} \mid t \rightarrow t$
  - $t1 \rightarrow t2$ is a the type of a function that, given an argument of type $t1$, returns a result of type $t2$
    - $t1$ is the *domain*, and $t2$ is the *range*

Type Judgments

- Our type system will prove *judgments* of the form
  - $A \vdash e : t$
  - "In type environment A, expression e has type t"

Type Environments

- A *type environment* is a map from variables to types (a kind of symbol table)
  - $\varnothing$ is the empty type environment
    - A closed term e is *well-typed* if $\varnothing \vdash e : t$ for some t
    - We'll abbreviate this as $\vdash e : t$
  - $A, x{:}t$ is just like A, except x now has type t
    - The type of x in $A, x{:}t$ is t
    - The type of $z{\neq}x$ in $A, x{:}t$ in the type of z in A
- When we see a variable in a program, we look in the type environment to find its type

Type Rules

$$\frac{}{A \vdash n : \text{int}} \qquad \frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \rightarrow t'} \qquad \frac{A \vdash e1 : t \rightarrow t' \quad A \vdash e2 : t}{A \vdash e1\ e2 : t'}$$

Example

$A = - : int \to int$

$$\dfrac{\dfrac{- \in dom(A)}{A \vdash - : int \to int} \qquad A \vdash 3 : int}{A \vdash -\ 3 : int}$$

Another Example

$A = + : int \to int \to int$
$B = A, x : int$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+ \in dom(B)}{B \vdash + :} \qquad \dfrac{x \in dom(B)}{B \vdash x : i}}{B \vdash +\ x : int \to int} \qquad B \vdash 3 : int}{B \vdash +\ x\ 3 : int}}{A \vdash (\lambda x{:}int.+\ x\ 3) : int \to int} \qquad A \vdash 4 : int}{A \vdash (\lambda x{:}int.+\ x\ 3)\ 4 : int}$$

We'd usually use infix x + 3

An Algorithm for Type Checking

- Our type rules are deterministic
  - For each syntactic form, only one possible rule
- They define a natural type checking algorithm
  - TypeCheck : type env × expression → type

TypeCheck(A, n) = int
TypeCheck(A, x) = if x in dom(A) then A(x) else fail
TypeCheck(A, λx:t.e) = TypeCheck((A, x:t), e)
TypeCheck(A, e1 e2) =
  let t1 = TypeCheck(A, e1) in
  let t2 = TypeCheck(A, e2) in
    if dom(t1) = t2 then range(t1) else fail

Semantics

- Here is a small-step, call-by-value semantics
  - If an expression can't be evaluated any more and is not a value, then it is *stuck*

$$\dfrac{}{(\lambda x.e1)\ v2 \to e1[v2\backslash x]} \qquad \dfrac{e1 \to e1'}{e1\ e2 \to e1'\ e2}$$

$$\dfrac{e2 \to e2'}{v1\ e2 \to v1\ e2'}$$

$e ::= v \mid x \mid e\ e$

$v ::= n \mid \lambda x{:}t.e$ *values* – not evaluated

Progress

- Suppose $\vdash e : t$. Then either e is a value, or there exists e' such that $e \to e'$
- Proof by induction on e
  - Base cases n, $\lambda x.e$ – these are values, so we're done
  - Base case x – can't happen (empty type environment)
  - Inductive case $e1\ e2$ – If $e1$ is not a value, then by induction we can evaluate it, so we're done, and similarly for $e2$. Otherwise both $e1$ and $e2$ are values. Inspection of the type rules shows that $e1$ must have a function type, and therefore must be a lambda since it's a value. Therefore we can make progress.

Preservation

- If $\vdash e : t$ and $e \to e'$ then $\vdash e' : t$
- Proof by induction on $e \to e'$
  - Induction (easier than the base case!). Expression e must have the form $e1\ e2$.
  - Assume $\vdash e1\ e2 : t$ and $e1\ e2 \to e'$. Then we have $\vdash e1 : t' \to t$ and $\vdash e2 : t'$.
  - Then there are three cases.
    - If $e1 \to e1'$, then by induction $\vdash e1 : t' \to t$, so $e1'\ e2$ has type t
    - If reduction inside $e2$, similar

Preservation, cont'd

- Otherwise $(\lambda x.e)\ v \to e[v\backslash x]$. Then we have

$$\frac{x : t' \vdash e : t}{\vdash \lambda x.e : t' \to t}$$

  - Thus we have
    - $x : t' \vdash e : t$
    - $\vdash v : t'$
  - Then by the substitution lemma (not shown) we have
    - $\vdash e[v\backslash x] : t$
  - And so we have preservation

Substitution Lemma

- If $A \vdash v : t$ and $A, x{:}t \vdash e : t'$, then $A \vdash e[v\backslash x] : t'$
- Proof: Induction on the structure of e
- For lazy semantics, we'd prove
  - If $A \vdash e1 : t$ and $A, x{:}t \vdash e : t'$, then $A \vdash e[e1\backslash x] : t'$

Soundness

- So we have
  - Progress: Suppose $\vdash e : t$. Then either e is a value, or there exists e' such that $e \to e'$
  - Preservation: If $\vdash e : t$ and $e \to e'$ then $\vdash e' : t$
- Putting these together, we get soundness
  - If $\vdash e : t$ then either there exists a value v such that $e \to^* v$, or e diverges (doesn't terminate).
- What does this mean?
  - Evaluation getting stuck is bad, so
  - "Well-typed programs don't go wrong"

Product Types (Tuples)

$e ::= \ldots \mid (e, e) \mid \text{fst } e \mid \text{snd } e$

$$\frac{A \vdash e1 : t \quad A \vdash e2 : t'}{A \vdash (e1, e2) : t \times t'}$$

$$\frac{A \vdash e : t \times t'}{A \vdash \text{fst } e : t} \qquad \frac{A \vdash e : t \times t'}{A \vdash \text{snd } e : t'}$$

- Or, maybe, just add functions
  - $\text{pair} : t \to t' \to t \times t'$
  - $\text{fst} : t \times t' \to t$
  - $\text{snd} : t \times t' \to t'$

Sum Types (Tagged Unions)

$e ::= \ldots \mid \text{inL}_{t2}\ e \mid \text{inR}_{t1}\ e$

$\quad \mid (\text{case } e \text{ of } x1{:}t1 \to e1 \mid x2{:}t2 \to e2)$

$$\frac{A \vdash e : t1}{A \vdash \text{inL}_{t2}\ e : t1 + t2} \qquad \frac{A \vdash e : t2}{A \vdash \text{inR}_{t1}\ e : t1 + t2}$$

$$\frac{\begin{array}{c} A \vdash e : t1 + t2 \\ A, x1{:}t1 \vdash e1 : t \quad A, x2{:}t2 \vdash e2 : t \end{array}}{A \vdash (\text{case } e \text{ of } x1{:}t1 \to e1 \mid x2{:}t2 \to e2) : t}$$

Self Application and Types

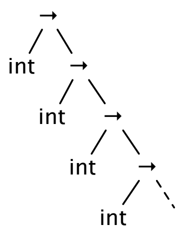
- Self application is not checkable in our system

$$\cfrac{\cfrac{A, x{:}? \vdash x : t \to t' \qquad A, x{:}? \vdash x : t}{A, x{:}? \vdash x\ x : \ldots}}{A \vdash \lambda x{:}?.x\ x : \ldots}$$

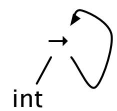
  - It would require a type t such that $t = t \to t'$
    - (We'll see this next, but so far...)
- The simply-typed lambda calculus is *strongly normalizing*
  - Every program has a normal form
  - I.e., every program halts!

Recursive Types

- We can type self application if we have a type to represent the solution to equations like $t = t \to t'$
  - We define the type $\mu\alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu\alpha.\text{int} \to \alpha$

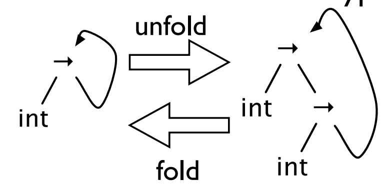


or



Folding and Unfolding

- We can check type equivalence with the previous definition
  - Standard unification, omit occurs checks
- Alternative solution:
  - The programmer puts in explicit *fold* and *unfold* operations to expand/contract one "level" of the type trees
    - $\text{unfold}\ \mu\alpha.t = t[\mu\alpha.t \backslash \alpha]$
    - $\text{fold}\ t[\mu\alpha.t \backslash \alpha] = \mu\alpha.t$



Fold-based Recursive Types

- $e ::= \ldots \mid \text{fold } e \mid \text{unfold } e$

$$\frac{A \vdash e : t[\mu\alpha.t \backslash \alpha]}{A \vdash \text{fold } e : \mu\alpha.t} \qquad \frac{A \vdash e : \mu\alpha.t}{A \vdash \text{unfold } e : t[\mu\alpha.t \backslash \alpha]}$$

ML Datatypes

- Combines fold/unfold-style recursive and sum types
  - Each occurrence of a type constructor when producing a value corresponds to occurrences of inL/inR and, when recursion is involved, fold
  - Each occurrence of a type constructor in a pattern match corresponds to a case and, when recursion is involved, (at least one) unfold

ML Datatypes Example

- type list = Int of int | Cons of int * int list
  - Equivalent to $\mu\alpha.\text{int}+(\text{int} \times \alpha)$
- (Int 3) equivalent to
  - fold $(\text{inL}_{\text{int}\times\mu\beta.\text{int}+(\text{int}\times\beta)}\ 3)$
- (Cons (2,(Int 3)) equivalent to
  - fold $(\text{inR}_{\text{int}}\ (2, \text{fold}\ (\text{inL}_{\text{int}\times\mu\beta.\text{int}+(\text{int}\times\beta)}\ 3)))$
- match e with Int x -> e1 | Cons x -> e2 same as
  - case (unfold e)
    - x:int → e1
    - | x: int×(μβ.int+(int×β)) → e2

Discussion

- In the pure lambda calculus, every term is typable with recursive types
  - (Pure = variables, functions, applications only)
- Most languages have some kind of "recursive" type
  - E.g., for data structures like lists, tree, etc.
- However, usually two recursive types that define the same structure but use a different name are considered different
  - E.g., struct foo { int x; struct foo *next; } is different from struct bar { int x; struct bar *next; }

Recap

- We've discussed simple types so far
  - Integers, functions, pairs, unions
  - Extensions for recursive types and updatable refs
- Type systems have nice properties
  - Type checking is straightforward (needs annotations)
  - Well typed programs don't go "wrong"
    - They don't get stuck in the operational semantics
- But...We can't type check all good programs

Up Next: Improving Types

- How can we build more flexible type systems?
  - More programs type check
  - Type checking is still tractable
- How can reduce the annotation burden?
  - Type inference

Parametric Polymorphism

- Observation: $\lambda x.x$ returns its argument exactly and places no constraints on the type of x
  - The identity function works for any argument type
- We can express this with *universal quantification:*
  - $\lambda x.x : \forall\alpha.\alpha\to\alpha$
  - For any type α, the identity function has type $\alpha\to\alpha$
  - This is also known as *parametric polymorphism*

System F: annotated polymorphism

- Let's extend our system as follows:
  - $t ::= \alpha \mid int \mid t \to t \mid \forall\alpha.t$
  - $e ::= n \mid x \mid \lambda x.e \mid e\ e \mid \Lambda\alpha.e \mid e\ [t]$
- That is, we add polymorphic types, and we add explicit *type abstraction (generalization)* …
  - Annotated code locations at which a value of polymorphic type is created
- … and *type application (instantiation)*
  - Explicitly annotated code locations at which a value of polymorphic type is used
- This system due to Girard, concurrently Reynolds

Defining Polymorphic Functions

- Polymorphic functions map types to terms
  - Normal functions map terms to terms
- Examples
  - $\Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha\to\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.y : \forall\alpha.\forall\beta.\alpha\to\beta\to\beta$

Instantiation

- When we use a parametric polymorphic type, we *apply* (or *instantiate*) it with a particular type
  - In System F this is done by hand:
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t1] : t1 \to t1$
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t2] : t2 \to t2$
- This is where the term *parametric* comes from
  - The type $\forall\alpha.\alpha\to\alpha$ is a "function" in the domain of types, and it is passed a parameter at instantiation time

Type Rules

$$\frac{A,\alpha \vdash e : t}{A \vdash \Lambda\alpha.e : \forall\alpha.t} \qquad \frac{A \vdash e : \forall\alpha.t}{A \vdash e[t'] : t[t'\backslash\alpha]}$$

- Notice that there are no constructs for manipulating values of polymorphic type
  - This justifies instantiation with *any* type—that's what the forall means!
- Note also that we are adding α to A; we could (should?) use this to ensure types are well-formed

Small-step Semantics Rules

$$\frac{}{(\Lambda\alpha.e)[t] \to e[t\backslash\alpha]}\text{(type-app)} \qquad \frac{e \to e'}{e[t] \to e'[t]}\text{(tapp-cong)}$$

- We have to extend substitution to include types; that's up next … !

Free Variables, Again

- We're going to need to perform substitutions on quantified types
  - So just like with lambda calculus, we need to worry about free variables and capture-free substitution
- Define the free variables of a type
  - $FV(\alpha) = \{\alpha\}$
  - $FV(c) = \varnothing$
  - $FV(t \to t') = FV(t) \cup FV(t')$
  - $FV(\forall\alpha.t) = FV(t) - \{\alpha\}$

Substitution, Again

- Define $t[u\backslash\alpha]$ as
  - $\alpha[u\backslash\alpha] = u$
  - $\beta[u\backslash\alpha] = \beta$ where $\beta \neq \alpha$
  - $(t \to t')[u\backslash\alpha] = t[u\backslash\alpha] \to t'[u\backslash\alpha]$
  - $(\forall\beta.t)[u\backslash\alpha] = \forall\beta.(t[u\backslash\alpha])$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
- Define $e[u\backslash\alpha]$ as
  - $(\lambda x{:}t.e)[u\backslash\alpha] = \lambda x{:}t[u\backslash\alpha].e[u\backslash\alpha]$
  - $(\Lambda\beta.e)[u\backslash\alpha] = \Lambda\beta.e[u\backslash\alpha]$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
  - $(e1\ e2)[u\backslash\alpha] = e1[u\backslash\alpha]\ e2[u\backslash\alpha]$
  - $x[u\backslash\alpha] = x$ and $n[u\backslash\alpha] = n$

Type Inference

- Let's consider the simply typed lambda calculus with integers
  - $e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$
  - (No parametric polymorphism)
- *Type inference*: Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?

Type Language

- Problem: Consider the rule for functions

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \rightarrow t'}$$

- Without type annotations, where do we get t?
  - We'll use *type variables* to stand for as-yet-unknown types
    - $t ::= \alpha \mid int \mid t \rightarrow t$
  - We'll generate *equality constraints* $t = t$ among the types and type variables
    - And then we'll solve the constraints to compute a typing

Type Inference Rules

$$\frac{}{A \vdash n : int} \qquad \frac{x \in dom(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}\alpha \vdash e : t' \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \rightarrow t'} \qquad \frac{A \vdash e1 : t1 \quad A \vdash e2 : t2 \quad \boxed{t1 = t2 \rightarrow \beta} \quad \beta \text{ fresh}}{A \vdash e1\ e2 : \beta}$$

"Generated" constraint

Example

$$\frac{\dfrac{A, x{:}\alpha \vdash x{:}\alpha}{A \vdash (\lambda x.x) : \alpha \rightarrow \alpha} \quad A \vdash 3 : int \quad \alpha \rightarrow \alpha = int \rightarrow \beta}{A \vdash (\lambda x.x)\ 3 : \beta}$$

- We collect all constraints appearing in the derivation into some set C to be solved
- Here, C contains just $\alpha \rightarrow \alpha = int \rightarrow \beta$
  - Solution: $\alpha = int = \beta$
- Thus this program is typable, and we can derive a typing by replacing α and β by int in the proof

Solving Equality Constraints

- We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set
  - $C \cup \{int=int\} \Rightarrow C$
  - $C \cup \{\alpha=t\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t=\alpha\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t1\to t2=t1'\to t2'\} \Rightarrow C \cup \{t1=t1'\} \cup \{t2=t2'\}$
  - $C \cup \{int=t1\to t2\} \Rightarrow$ unsatisfiable
  - $C \cup \{t1\to t2=int\} \Rightarrow$ unsatisfiable

Termination

- We can prove that the constraint solving algorithm terminates.
- For each rewriting rule, either
  - We reduce the size of the constraint set
  - We reduce the number of "arrow" constructors in the constraint set
- As a result, the constraint always gets "smaller" and eventually becomes empty
  - A similar argument is made for strong normalization in the simply-typed lambda calculus

Occurs Check

- We don't have recursive types, so we shouldn't infer them
- So in the operation $C[t\backslash\alpha]$, require that $\alpha \notin FV(t)$
  - (Except if t = a, in which case there's no recursion in the types, so unification should succeed)
- In practice, it may better to allow $\alpha \in FV(t)$ and do the occurs check at the end
  - But that can be awkward to implement

Unifying a Variable and a Type

- Computing $C[t\backslash\alpha]$ by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
  - The terms are in a union-find forest
  - When a variable and a term are equated, we union them so they have the same ECR (equivalence class representative)
    - Want the ECR to be the concrete type with which variables have been unified, if one exists. Can read off solution by reading the ECR of each set.

Example

$\alpha = int \rightarrow \beta$

$\gamma = int \rightarrow int$

$\alpha = \gamma$

Unification

- The process of finding a solution to a set of equality constraints is called *unification*
  - Original algorithm due to Robinson
    - But his algorithm was inefficient
  - Often written out in different form
    - See Algorithm W
  - Constraints usually solved on-line
    - As type inference rules applied

Discussion

- The algorithm we've given finds the *most general type* of a term
  - Any other valid type is "more specific," e.g.,
    - $\lambda x.x : int \rightarrow int$
  - Formally, any other valid type can be gotten from the most general type by applying a substitution to the type variables
- This is still a *monomorphic* type system
  - α stands for "some particular type, but it doesn't matter exactly which type it is"

Inference for Polymorphism

- We would like to have the power of System F, and the ease of use of type inference
  - In short: given an untyped lambda calculus term, can we discover the annotations necessary for typing the term in System F, if such a typing is possible?
  - Unfortunately, no. This problem has been shown to be undecidable.
- Can we at least perform some type inference for parametric polymorphism?
  - Yes. A sweet spot was found by Hindley and Milner
  - But first, let's consider the general case …

Attempting Type Inference

- Let's extend simply-typed calculus as follows:
  - $t ::= \alpha \mid \text{int} \mid t \to t \mid \forall\alpha.t$
  - $e ::= n \mid x \mid \lambda x.e \mid e\ e$
- Type inference will automatically infer where to generalize a term, to introduce polymorphic types, and where to instantiate them

Instantiation

$$\frac{A \vdash e : \forall\alpha.t}{A \vdash e : t[t'\backslash\alpha]}$$

- This rule is exacty the same as System F, but we just "magically" pick which t' to instantiate with
  - You're surely wondering about algorithmics. We'll get to that …

Generalization

- Question: When is it safe to generalize (quantify) a type variable α in the type of expression e?
- Answer: Whenever we can redo the typing proof for e, choosing α to be anything we want, and still have a valid typing proof.

Examples

$$\frac{A, x{:}\alpha \vdash e : \alpha}{A \vdash \lambda x.x : \alpha \to \alpha}$$

$$\frac{A, x{:}\text{int} \vdash x : \text{int}}{A \vdash \lambda x.x : \text{int} \to \text{int}}$$

$$\frac{A, x{:}(i \to i) \vdash x : (i \to i)}{A \vdash \lambda x.x : (i \to i) \to (i \to i)}$$

- The choice of the type of x is purely local to type checking $\lambda x.x$
  - There is no interaction with the outside environment
  - Thus we can generalize the type of x

Examples (cont'd)

$$\frac{A, x{:}int \vdash x : int}{A \vdash \lambda x.x+3 : int \to int}$$

- The function restricts the type of x, so we cannot introduce a type variable
  - Thus we cannot generalize the type of x
  - We can only generalize when the function doesn't "look at" its parameter

Examples (cont'd)

$$\frac{A, y{:}\alpha,\ x{:}\alpha \vdash \text{if } p \text{ then } x \text{ else } y : \alpha}{A, y{:}\alpha \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y :}$$

$$\Downarrow$$

$$\cancel{\frac{A, y{:}\alpha,\ x{:}int \vdash \text{if } p \text{ then } x \text{ else } y : int}{A, y{:}\alpha \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y : int \to int}}$$

- The choice of the type of x depends on the type environment
  - In the first derivation, x and y have the same type; if we generalize the type of x, they could have different types
  - Thus we cannot generalize the type of x

Generalization Rule

$$\frac{A \vdash e : t \qquad \alpha \notin FV(A)}{A \vdash e : \forall\alpha.t}$$

- We can generalize any type variable that is unconstrained by the environment
  - Warning: This won't quite work with refs

Another Justification

- Suppose we have
  - $A \vdash e : t$ and $\alpha \notin FV(A)$
- Then let u be any type. By induction, can show
  - $A[u\backslash\alpha] \vdash e : t[u\backslash\alpha]$
  - But then since $\alpha \notin FV(A)$, that's equivalent to
  - $A \vdash e : t[u\backslash\alpha]$

Polymorphic Type Inference

- We'd like to extend our algorithm to polymorphic type inference
  - Performance generalization and instantiation automatically (and deterministically)
- Major problem: Our system for polymorphism is too expressive

Hindley-Milner Polymorphism

- Restrict polymorphism to only the "top level"
  - Introduce polymorphism at let
  - Fully instantiate at use of a polymorphic type
- Here is our new language
  - e ::= n | x | λx.e | e e | let x = e in e
  - t ::= α | int | t → t
  - s ::= t | ∀α.s
    - These are *type schemes*
  - A ::= ∅ | A,x:s
  - Notice that, according to the prior instantiation rule, we won't instantiate α with a scheme s, only a type t

Old Type Inference Rules

$$\frac{}{A \vdash n : \mathrm{int}}$$

$$\frac{A, x{:}\alpha \vdash e : t' \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t'}$$

$$\frac{A \vdash e1 : t1 \quad A \vdash e2 : t2 \quad t1 = t2 \to \beta \quad \beta \text{ fresh}}{A \vdash e1\ e2 : \beta}$$

New Type Inference Rules

- At let, generalize over all possible variables

$$\frac{A \vdash e1 : t1 \quad A, x{:}\forall\vec{\alpha}.t1 \vdash e2 : t2 \quad \vec{\alpha} = FV(t1) - FV(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

- At variable uses, instantiate to all fresh types

$$\frac{A(x) = \forall\vec{\alpha}.t \quad \vec{\beta} \text{ fresh}}{A \vdash x : t[\vec{\beta} \backslash \vec{\alpha}]}$$

- Here the $\vec{\alpha}$ denotes a list of type variables

Algorithm W

- A type inference algorithm that explicitly solves the equality constraints on-line
- Instead of implicit global substitution (like we used before), threads the substitution through the inference
- In practice, use previous algorithm, plus generalize at let and instantiate at variable uses.
  - Solve for the type of e1, generalize it, then instantiate its solution when doing inference on e2

Example

- Parametric polymorphic type inference

  let x = λx.x in // x : $\forall\alpha.\alpha\rightarrow\alpha$

  x 3; // x : $\beta\rightarrow\beta$, β=int

  x (λy.y) // x : $\gamma\rightarrow\gamma$, $\gamma=\delta\rightarrow\delta$

- This would be untypable in a monomorphic type system

Kinds of Polymorphism

- We've just seen parametric polymorphism
  - System F and Hindley-Milner style polymorphism
- Another popular form is subtype polymorphism
  - As in OO programming
  - These two can be combined (e.g., Java Generics)
- Some languages also have *ad-hoc polymorphism*
  - E.g., + operator that works on ints and floats
  - E.g., overloading in Java

An Imperative Language

e ::= x | λx.e | e e	
| ref e	allocation
| !e	dereference
| e := e	assignment
| e; e	sequencing

- Notice that this is not C
  - Variables cannot be updated; only references can
  - I.e., there are no l-values or r-values
- This is a language with *updatable references*

Examples

!(ref 0)

let x = ref 0 in
  x := !x + 1

let x = ref 0 in
  λy. x := !x + 1; !x

Type Checking Rules

- t ::= ... | ref t
  - Note: in ML this type is written t ref

$$\frac{A \vdash e : t}{A \vdash \text{ref } e : \text{ref } t} \qquad \frac{A \vdash e : \text{ref } t}{A \vdash \,!e : t}$$

$$\frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : t}$$

Unit and the Unit Type

- Sometimes in imperative programs we write expressions that have some *side effect* but no interesting result
- To represent this directly, use *unit:*
  - e ::= ... | ()
  - t ::= ... | unit

$$\frac{}{A \vdash () : \text{unit}} \qquad \frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : \text{unit}}$$

Operational Semantics

- Now we need to keep track of memory
  - State is a map from locations to values
  - Our redexes will be tuples ‹State, expression›
  - As a consequence, order of evaluation matters
- As before, evaluation will yield a fully-evaluated term, also called a *value*
  - v ::= x | λx.e
  - e ::= v | e e | ref e | !e | e := e

Operational Semantics (cont'd)

$$\frac{}{\langle S, (\lambda x.e1)\rangle \to \langle S, (\lambda x.e1)\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', v1\rangle \quad \langle S', e2\rangle \to \langle S'', v2\rangle}{\langle S, e1; e2\rangle \to \langle S'', v2\rangle}$$

$$\frac{\langle S, e\rangle \to \langle S', v\rangle \qquad \text{loc fresh}}{\langle S, \text{ref } e\rangle \to \langle S[v\backslash loc], loc\rangle}$$

Operational Semantics (cont'd)

$$\frac{\langle S, e\rangle \to \langle S', loc\rangle}{\langle S, !e\rangle \to \langle S', S'(loc)\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', loc\rangle \qquad \langle S', e2\rangle \to \langle S'', v\rangle}{\langle S, e1 := e2\rangle \to \langle S''[v\backslash loc], v\rangle}$$

$$\frac{\langle S, e1\rangle \to \langle S', \lambda x.e\rangle \quad \langle S', e2\rangle \to \langle S'', v\rangle \quad \langle S'', e[v\backslash x]\rangle \to \langle S''', v'\rangle}{\langle S, e1\ e2\rangle \to \langle S''', v'\rangle}$$

Polymorphism and References

- Suppose we want polymorphism in our imperative language
  - e ::= x | n | λx.e | e e | ref e | !e | e := e
  - s ::= t | ∀α.s
  - t ::= α | int | t → t | ref t
- What if we try our standard rule?

$$\frac{A \vdash e1 : t1 \qquad A, x{:}\forall\vec{\alpha}.t1 \vdash e2 : t2 \qquad \vec{\alpha} = FV(t1) - FV(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

Naive Generalization is Unsound

- Example (due to Tofte)

  let r = ref (λx.x) in // r : ∀α.ref (α→α)

  r := λx.x+1; // checks; use r at ref (int → int)

  (!r) true // oops! checks; use r at ref(bool →bool)

- α should not be generalized, because later uses of r may place constraints on it
- Nobody realized there was a problem for a long time

Solution: The Value Restriction

- Only allow *values* to be generalized
  - $v ::= x \mid n \mid \lambda x.e$
  - $e ::= v \mid e\ e \mid \text{ref}\ e \mid !e \mid e := e$

$$\frac{A \vdash v : t1 \qquad A, x{:}\forall\alpha.t \vdash e2 : t2 \qquad \alpha = FV(t) - FV(A)}{A \vdash \text{let } x = v \text{ in } e2 : t2}$$

  - Intuition: Values cannot later be updated
  - This solution due to Wright and Felleisen
    - Tofte found a much more complicated solution

Benefits of Type Inference

- Handles higher-order functions
- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forward/backward distinction
- Polymorphism provides context-sensitivity

Drawbacks to Type Inference

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand
- Polymorphic type inference may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)

Drawbacks to Type Inference

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand
- Polymorphism may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)